Master in High Performance Computing

Porting of the LBE3D to GPU with OpenACC

Supervisor(s):

Ivan Girotto,

Federico Toschi

Candidate:

Saeid Aliei

6th edition
2019–2020

Abstract

With the increasing demand of computational capability at low-power, graphic accelerators are today dominating not only the gaming market but also equipping the majority of the most powerful supercomputer infrastructures worldwide. On the other hand, the enabling on those accelerated systems of complex scientific applications has not followed the technological shift, but resulted particularly challenging and requiring significant effort in writing codes on specific and, usually not portable, languages.

Refactoring large code-base applications requires man power and usually is not portable and prone to errors. Among the several attempts to develop directive based languages to port applications for heterogenous systems, OpenACC has become among the most promising paradigms. This work is an attempt to port on a world-class multi-GPU distributed hybrid system, as the Marconi-100 hosted at CINECA, a production-ready, single-relaxation time, multi-component Lattice-Boltzamann Methods (LBM) based application, using OpenACC. It includes a detailed performance analysis of the application, also with a comparison regards previous results obtained on computer platforms equipped with x64-86 based Intel computer platform for high-end computing.

Acronyms

AoS Array of Structures. 27

BCs Boundry Conditions. 3

BGK Bhatnagar-Gross-Krook. 10, 11

BSC Barcelona Supercomputing Center. 41

CAoSoA Clustered Array of Structure of Arrays. 28

CFD Computational Fluid Dynamics. 3, 27

CSoA Clustered Structure of Arrays. 28

FF Fully Fused. 39

GPGPU General Purpose computing on Graphics Processing Units. 2

GPU Graphics Processing Unit. 1, 2, 13, 19, 27, 36, 39

GPUs Graphics Processing Unit. 35

KNL Knights Landing. 28, 41

LBE3D Lattice-Boltzamann Equation in 3D. iv, v, 1, 2, 9, 10, 27, 28, 33, 35, 47

LBM Lattice-Boltzamann Methods. i, 2–4, 10, 26, 27, 38, 47

MIC Many Integrated Cores. 27, 41

MPI Message Passing Interface. 36

PBC Periodic Boundry Condition. 3

SIMS Single Instruction Multiple Data. 27, 28

SIMT Single Instruction Multiple Threads. 14, 28

SKL SkyLake. 41, 47

SoA Structure of Arrays. 27, 28, 39

TLB Translation Lookaside Buffer. 28

Contents

List of Figures

Chapter 1

Introduction

In the past decade heterogenous systems has become the de facto processing units for large scale simulations, as evident from the Top500[7] list 8 out of 10 currently most powerful HPC systems are hybrid systems, that take advantage of accelerators, mostly Graphics Processing Unit (GPU) vendored by NVIDIA®. Designing efficient, maintainable and portable applications for heterogeneous systems has been usually a cumbersome effort, since these accelerators use a specific programming language such as CUDA®. In the recent years OpenACC has emerged as the new standard in compiler directive based interfaces for portable and unified, accelerator programming. There has been a skew of scientific applications efficiently offloading their applications to accelerators. In this work we present the OpenACC implementation of an LBM based application, named LBE3D. Extending from previous works, we start analyzing several data-layouts designed for highly efficient LBM based applications. The analysis is meant to select the most optimal configuration for hybrid CPU/GPU systems. Based on the results of this effort we perform the porting of the LBE3D, with a detailed performance analysis.

We start by introducing the theoretical foundations of the numerical models implemented in the LBE3D, and then we are going to introduce the

GPU architecture and the programming models used to write applications for General Purpose computing on Graphics Processing Units (GPGPU). We describe main features of the OpenACC paradigm, and discuss whether is suited for an affordable, portable, and maintainable offloading of scientific applications to GPUs. After that we are going to review the LBE3D optimization, and the porting of the application to GPUs with OpenACC directives. We show that our porting results efficient, if compared with performance data obtained on high-end Intel platform during past productions, and scalable on distributed multi-GPU compute nodes.

This thesis is outlined as such. In Chapter 2 we are going to review the LBM theory and background, including a brief introduction of the Poiseuille flow case, used to benchmark the different data-layouts, other then as validation ground for the application. In Chapter 3 we are going to first introduce the GPGPU architecture and its massively data parallel capabilities, including a review of two programming models, namely CUDA and OpenACC. At last we are going to show the results we achieved with the porting of Poiseulle and multi-component on both single- and multi-GPU distributed platforms.

Chapter 2

Lattice-Boltzmann Methods

In this chapter we are going to discuss the theoretical and computational aspects of the Lattice Boltzamann Equations and the resulting base algorithm that has had tremendous effect on engeering problems and doing fundamental science. In the last two decade or so, LBM has emerged as a promising tool for modeling the Navier-Stokes equations and simulation of complex fluids flows such as porus media flows or in other fields of science, in biomedical flows, earth science(soil filteration), Energy Sciences(fuel cells) and so on. Our focus in here will be the most important utilization of the LBM, which is in Computational Fluid Dynamics (CFD). LBM is based on microscopic models and mesoscopic kinetic equations. The Lattice Boltzmann Methods can be viewed as finite difference method for solving the Boltzmann transport equation and we can also recover the Navier-Stokes equations with LBM by choosing a proper collision operator.

We are going to first introduce LBM and the base algorithm beneath it, then we will introduce the commonly used Boundry Conditions (BCs), then using a Periodic Boundry Condition (PBC) we are going to drive the analytical solution for a steady plane Poiseuille flow, and then compare the velocity profile with the simulations of the LBE3D application.

In section 2 we are going to introduce the Multi-component flows and in doing so we are going to look at the basics of the Schen-Chen multicomponent model.

2.1 Lattice Boltzmann Methods

The Lattice-Boltzamann Methods (LBM) originates from the Boltzmann's kinetic theory of gases. The basic idea is that we can imagine fluids consisting of billions of particles, modelled by hard spheres that move randomly and have collisions, according to the general Boltzmann transport eqaution:

$$\frac{\partial f}{\partial t} + \vec{u}.\vec{\nabla} f + \vec{F}_{\text{ext}}.\vec{\nabla}_{\vec{u}} f = \Omega \tag{2.1}$$

Where $f(\vec{x}, t)$ is the particle distribution function, $\vec{u}$ is the particle velocity, $\vec{F}_{\text{ext}}$ is some external force, which we are going to neglect for now and Ω is the collision operator, which deals with all the collisions in the system that results in thermalizing the system. The LBM simplifies the computational effort of the Boltzmann's original idea by confining particles to the nodes of a lattice, where a particle can only move to it's neighbor and diagnoal nodes. For a 3 dimentional model a particle can only stream in a possible of 19 directions, including the stationary point. We refer to these velocities as the *microscopic velocities* and denote them with $\vec{e}_i$, where $i = 0, ..., 18$, as you can see in Fig.[**?**]. This model is commonly known as the D3Q19 and is the main model we will discuss here. There are of course other models, such as D3Q15, D3Q27 and D2Q9 for a two dimentional problem [18]. Fig. 2.1 shows a typical node of a D3Q19 with 19 veclocities, $\vec{e}_i$ defined by:

$$\vec{e_i} = \begin{cases} (0,0,0) & i = 0 \\ (\pm 1,0,0),(0,\pm 1,0),(0,0,\pm 1) & i = 1,...,6 \\ (\pm 1,\pm 1,\pm 1) & i = 7,...,18 \end{cases}$$

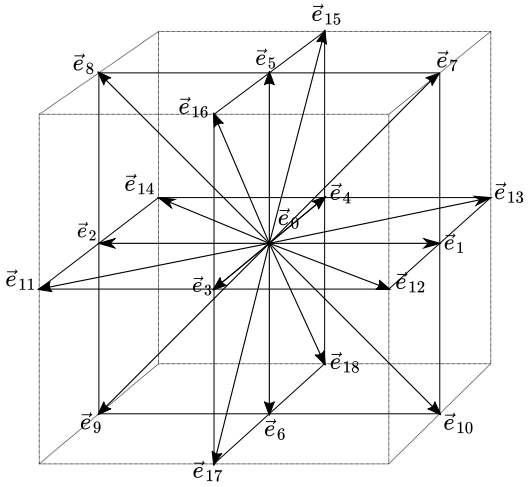

Figure 2.1: a D3Q19 Lattice node

For each particle on the lattice we associate a descrete probability distribution function $f_i(\vec{x_i}, \vec{e_i}, t)$ or simply $f_i(\vec{x}, t), i = 0, ..., 18$, which describes the probability of streaming in one particular direction of the lattice. The *macroscopic* fluid density can be defined as a summation of microscopic particle distribution function.

$$\rho(\vec{x}, t) = \sum_{i=0}^{18} f_i(\vec{x}, t) \tag{2.2}$$

or the mass flux can be defined as,

$$\vec{j}(\vec{x}, t) = \sum_{i=0}^{18} \vec{u_i} f_i(\vec{x}, t) \tag{2.3}$$

also the *macroscopic velocity*

$$\vec{u}(\vec{x},t) = \frac{1}{\rho}\sum_{i=0}^{18} c f_i \vec{e}_i \tag{2.4}$$

The key steps in LBM involve *Streaming* and *Collision* processes which are given by left and right hand side of the equation below, respectively:

$$f_i(\vec{x}+\vec{e}_i\Delta t, t+\Delta t) - f_i(\vec{x},t) = -\frac{1}{\tau}(f_i(\vec{x},t) - f_i^{\text{eq}}(\vec{x},t)) \tag{2.5}$$

In the implementing phase the streaming and collision steps are computed separatedly and for our application we do each phase in multiple kerenels.

In the collision term of equation 2.4, $f_i^{eq}(\vec{x},t)$ is the equilibrium eqaution,which is given by Boltzmann distribution, and τ is a measure of relaxation time towards the equilibrium. If we take an incompressible flow, from Maxwell-Boltzmann distribution we have:

$$\begin{aligned}
f_i^{eq}(\vec{x},t) &= \rho(\vec{x},t)\left(\frac{1}{2\pi RT}\right)^{\frac{d}{2}} \exp\left(\frac{-(\vec{e}_i-\vec{u})^2}{2\pi RT}\right) \\
&= \rho(\vec{x},t)\left(\frac{1}{2\pi RT}\right)^{\frac{d}{2}} \exp\left(\frac{-\vec{e}_i^2}{2\pi RT}\right)\exp\left(\frac{\vec{e}_i\vec{u}}{\pi RT} - \frac{\vec{u}^2}{2\pi RT}\right) \\
&= \rho(\vec{x},t)\left(\frac{1}{2\pi RT}\right)^{\frac{d}{2}} \exp\left(\frac{-\vec{e}_i^2}{2\pi RT}\right)\left(1 + \frac{\vec{e}_i\vec{u}}{RT} + \frac{(\vec{e}_i\vec{u})^2}{2(RT)^2} - \frac{\vec{u}^2}{2RT} + ...\right) \\
&= \omega_i\rho\left(1 + \frac{3\vec{e}_i\vec{u}}{c^2} + \frac{9(\vec{e}_i\vec{u})^2}{2c^4} - \frac{3(\vec{u})^2}{2c^2}\right) \\
&= \omega_i\rho + \rho s_i(\vec{u}(\vec{x},t))
\end{aligned}$$

in which $c = \sqrt{3RT}$ is the lattice sound speed and $s_i(\vec{u})$ is defined as,

$$s_i(\vec{u}) = \omega_i \left[3\frac{\vec{e}_i.\vec{u}}{c} + \frac{9}{2}\frac{(\vec{e}_i.\vec{u})^2}{c^2} - \frac{3}{2}\frac{\vec{u}.\vec{u}}{c^2} \right] \tag{2.6}$$

where ω_i is the weights,

$$\omega_i = \begin{cases} 1/3 & i = 0 \\ 1/18 & i = 1, ..., 6 \\ 1/36 & i = 7, ..., 18 \end{cases}$$

The physics of the fluid flows is mainly controlled by two dimensionless parameters, the Mach number and Reynolds number, respectively given by:

$$\mathrm{Ma} = \frac{u}{c}, \qquad \mathrm{Re} = \frac{uL}{\nu}$$

where ν is the kinematic viscosity, given by $\nu = \frac{\mu}{\rho}$ in which μ is the shear viscosity.

Now we can set the algorithm as follow:

Algorithm 1: Schematic Algorithm for LBM

Initialize ρ, u, f_i and f_i^{eq} ;
for $i \leftarrow 1$ **to** *NUM_STEPS* **do**
  $f_i^\star \leftarrow f_i$, streaming step in the direction of $\vec{e}_i$;
  Compute macroscopic quantities, ρ and $\vec{u}$ from $f_i^\star$ using equations 2.2 and 2.4 ;
  Compute f_i^{eq} using 2.1 ;
  Collision phase: update the distribution function according to $f_i = f_i^\star - \frac{1}{\tau}(f_i^\star - f_i^{eq})$ using 2.5
end

Most of the execution time is spent in streaming and collision operators. Collison kernel is compute bound and streaming kernel is memory bound which makes it ideal for performance evaulation of current pre-exa scale computatioal efforts.

2.1.1 Boundry Conditions

Boundary Conditions (BCs) are central to the statbility and the accuracy of any numerical solution. For the lattice Boltzmann method, the descrete distribution functions on the boundry has to be such that it reflects the macroscopic BCs imposed on the system. Moreover on the boundry nodes, the distribution function assigned to each vectors $\vec{e}_i$ pointing out of the lattice move out of the computatinal domain in the streaming step [12] and the ones assgined to the opposite vectors, are undefined because there is no node which the distributions could come from, therefore special rules must be applied for boundry nodes. These BCs can be chosen in various manners, periodic boundries are realized by streaming $\vec{f}_i$ leaving the computational domain on the one boundry to the boundry nodes located on the opposite side of the domain. For pressure driven, incompressable flow, the generalized periodicity conditions of the flow can be written as:

$$\vec{u}(\vec{x} + \Delta\vec{x}, t) = \vec{u}(\vec{x}, t)$$

2.1.2 Poiseuille Flow

Imagine a steady flow past a channel driven by a pressure gradient at the inlet and outlet of the channel, see Fig. 2.2, we are going to apply the lattice BGK model to simulate it and also we are going to derive the analytical solution for this simple case.

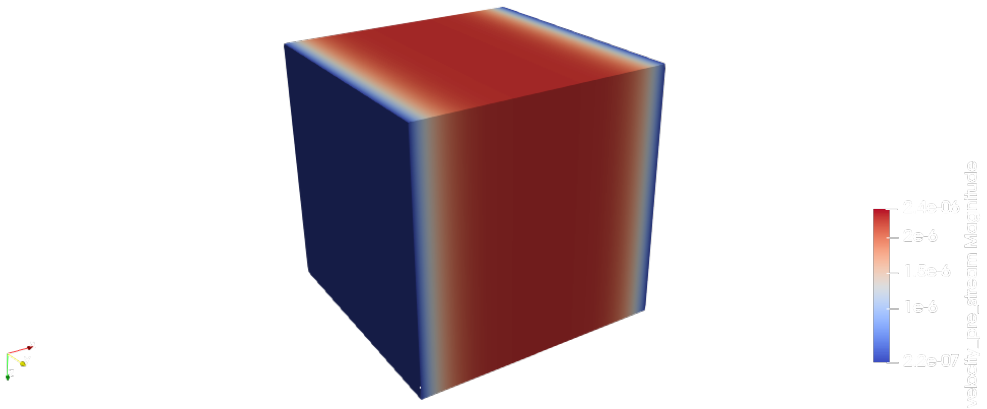

Figure 2.2: Plane Poiseuille Flow. This is the results from one of the runs of the LBE3D code.

By using symmetry and incompressibility, the velocity components u and v do not have any horizontal variations and $v \equiv 0$. The Navier-Stokes equations are further reduced to:

$$\mu \frac{\partial^2 u}{\partial y^2} = \frac{\partial p}{\partial x}, \tag{2.7}$$

Where $\frac{\partial p}{\partial x} = \frac{P_1 - P_0}{L}$. The initial and boundry conditions are:

$$\begin{aligned} u(x,y,0) = v(x,y,0) = 0, \qquad & p(x,y,0) = P_{avg} \\ u(x,0,t) = v(x,0,t) = 0, \qquad & u(x,H,t) = v(x,H,t) = 0 \\ p(0,y,t) = P_0, \qquad & p(L,y,t) = P_1 \end{aligned}$$

where $P_{avg} = (P_0 + P_1)/2$, and P_0 and P_1 are the pressure at the inlet and outlet of the channel. Then the steady Poiseuille flow has the exact solution for velocity as:

$$u(x,y,t) = \frac{\Delta p}{2\mu L} y(y-H)$$
$$v(x,y,t) = 0$$

See Fig. **??**, that shows the x component of the velocity distribution along y directin. The data points are gathered from one of the runs of LBE3D.

2.2 Multi-component Lattice Boltzmann models

There are number of models which take to account multiple fluid components, here we will discuss the *Shan-Chen Multi-componet* model [16, 22]. interested reader is encouraged to see the references for the *Free Energy* based models, e.g. [17] , *Colour Gradient*[10]lattice boltzmann models and optimized versions of it, e.g.[14].

2.2.1 The Shan-Chen pseudo potential model

We start with the standard LBM using the Bhatnagar-Gross-Krook (BGK) collision term [3]. We recall that in the LBM population of particles on lattice nodes are used to describe fluids. The distribution function $f_i(\vec{x}) = f(\vec{x}, \vec{e}_i)$ is used to describe the populations at each lattice site. From equation 2.5 we know that the evolution of the distribution function is governed by the BGK collision operator

$$f_i(\vec{x} + \vec{e}_i \Delta t, t + \Delta t) = f_i(\vec{x}, t) - \frac{1}{\tau}(f_i(\vec{x}, t) - f_i^{\mathrm{eq}}(\vec{x}, t)) + K_i \qquad (2.8)$$

To incorporate a body force $\vec{F}$, an extra term K_i is included in the BGK model. If we use the Guo's force term [11]:

$$\vec{K} = (1 - \frac{1}{2\tau})\omega_i[3\frac{\vec{e}_i - \vec{u}}{e^2} + 9\frac{\vec{e}_i.\vec{u}}{e^4}\vec{e}_i].\vec{F}, \tag{2.9}$$

The macroscopic velocity $\vec{u}$ is computed as,

$$\vec{u}(\vec{x}, t) = \sum_i \vec{e}_i f_i(\vec{x}, t) + \frac{\Delta t \vec{F}}{2}$$

Macroscopic transport equations for mass, momentum and energy can be derived from the Boltzmann equations using a Chapman-Enskog expansion [18]. The interaction force term between particles can be writen as,

$$\vec{F}(\vec{x}, t) = -G\psi_\sigma(\vec{x}, t) \sum_{i=1}^{8} \omega(|\vec{e}_i|)\psi_{\bar{\sigma}}(\vec{x} + \vec{e}_i\Delta t, t)\vec{e}_i, \tag{2.10}$$

where σ and $\bar{\sigma}$ denote the different fluid components. G is a parameter determining the interaction strength $F(\vec{x}, t)$ between neighbouring particles. It also determines whether the interaction is attractive or repulsive. To simulate a binary immiscible fluids system, the value of G should be kept positive so that a force will be generated to seperate the fluids away from the interface. ψ_σ is the effective number density which is taken as the component density, $\psi_\sigma = \rho_\sigma$. $\omega(|\vec{e}_i|)$ is a parameter related to the strength and order of isotropy of the interaction forces.

To incorporate the interaction force,Shan and Chen proposed a force term scheme which only shifts the equilibrium velocity. However this force scheme has been reported to be correct only if the relaxation time $\tau = 1$. This unfavorouble feature can be eliminated using Guo's force term [21].

Chapter 3

LBM based code on GPU using OpenACC

The current computing landscape is spotted with a variety of computing architecture, multi-core CPUs, GPUs, many-core devices, DSPs, and FPGAs, to name a few. It is now commonplace to find not just one, but several of these differing architectures within the same machine. Programmers must make portability of their code a forethought, otherwise they risk locking their application to a single architecture, which may limit their ability to run in future architectures. Although the variety of architectures may seem daunting to the programmer, closer analysis reveals trends that show a lot in common between them. The first thing to notice is that all of these architectures are moving in the direction of more parallelism. CPUs are not only adding CPU cores but also expanding the length of their SIMD operations. GPUs have grown to require a high degree of parallelism in order to achieve high performance. Modern processors need not only large amount of parallelism, but frequently expose multiple levels of parallelism with varying degrees of coarsness. The next thing to notice that all of these architectures have exposed hierarchies of memory. CPUs have the main system memory, typically DDR, and multiple layers of cache memory. GPUs have the main

CPU memory, the main GPU memory, and various degrees of cache memory. Moreover on hybrid architectures, where the two or more architectures have completely separate memories, some with physically separate but logically the same memory and some with fully shared memory[1]. Because of these complexities, it's important that developers choose a programming model that balances the need for portability with the need for performance.

3.1 Introduction to GPGPU

General Purpose Computing on GPUs is a realtively new phenomena. GPUs were first hardware blocks optimzed to do small graphics operations. Gaming industry among others pushed the development of flexible, programmable and more powerful GPUs, as demand grow for more flexible programmable GPUs, there were several research attemps to develop languages to prorgam GPUs more easily. In 2006 NVIDIA introduced CUDA architecture and it's data parallelism model, not suprisingly, fit well within data parallelism availabale in NVIDIA GPUs. GPU provides much higher instruction throughput and memory bandwidth than the CPU within a similar price and power envelope. Other computing devices, like FPGAs are also very energy efficient but offer much less programming flexibility than GPU[6].

3.1.1 An Overview of GPU Architecture

GPU architecture differes significantly from that of CPUs. The basic idea is simple, remove everything that makes a single instruction run fast, like cache, which makes up to 50% of die area of modern day CPUs, out-of-order control logic, complex branch predictions, memory prefetching, etc. Instead invest saved transistors into more copies of the simple core. The funding design assumption is, expect only data-parallel workloads and exploit this to maximum extent in the chip design. Here we will mention basic architecture

and concepts using an NVIDIA Volta architecture.
By using the concept of Single Instruction Multiple Threads (SIMT) NVIDIA GPUs have hundreds of cores that can process thousands of software threads simaltaneously. A GPU is connected to a host through a high-speed I/O bus, typically PCI-Express. Though newer generations use NVLink for more bandwidth and NVIDIA also recently introduced NVSwitch which will enable bandwidth of upto 900 GB/s betweeen GPUs in high-end HPC facilities. The GPU has it's own device memory, for a Tesla V100 it comes in 16 and 32 GB configuration. Data usually is transferred between the GPU and host memories using programmed DMA, which operates concurrently with the host and GPU compute units, though there is some support for direct access to host memeory form GPU under certain restrictions. As a GPU is designed for throughput computing or stream, it deos not depend on a deep cache memory hierarchy for memory performance. The device memory supports very high data bandwidth using a wide data path. On NVIDIA GPUs, it's 4096-bits wide for a V100, allowing 128 consecutive 32-bit words to be fetched from memory in a single cycle, which results in bandwidth of upto 900 GB/s. but this also means there is a severe effective bandwidth degradation for strided access. A stride-two access for instance will fetch those 4096 bits but only use half of them suffering a 50% bandwidth loss. NVIDIA GPUs have a number of multiprocessors, each of which executes in parallel with the others.
The GV100 inlcludes 21.1 billions transistors with a die size of $815mm^2$[2]. similar to previous generation GP100 GPU, the GV100 is composed of multiple Graphics Processing Clusters (GPCs), Texture Processing Cluster(TPCs), Streaming Multiprocessors(SMs), and memory controllers. A full GV100 is composed of 6 GPCs, 84 Volta SMs, 42 TPCs(each including two SMs), and eight 512-bit memory controllers(4096-bits total). Each SM has 64 INT32, 64 FP32, 32 FP64 Cores and 8 new Tensor cores. Each SM also includs four texture units. with 84 SMs, a full GV100 GPU has a total 5376 INT32 cores, 5376 FP32 cores, 2688 FP64 cores, 672 Tensor cores which are purpose built for deep learning workloads, and 336 texture units. Each memory controller is attached to 768KB of L2 cache, and each HBM2 DRAM stack is controlled

Figure 3.1: A Full NVIDIA GV100 Architecture

by a pair of memory controllers, for a total of 900 GB/s bandwidth, at 877 MHz, the full GV100 GPU includes a total of 6144KB of L2 cache. Fig3.2 shows a full GV100 GPU.

The Tesla V100 accelerator uses 80 SMs, which can deliver peak performances of[15]:

- 7.8 TFLOP/s of double precision floating-point (FP64) performance.
- 15.7 TFLOP/s of double precision floating-point (FP32) performance.
- 125 TFLOPS/s of mixed precision matrix-multiply-and-accumulate.

The block diagram of an SM is shown at Fig.3.2

The volta SM is partioned into 4 processing blocks, instruction from the same warp are allocated to a specific scheduler processing block and can only ac-

Figure 3.2: A Full NVIDIA GV100 Architecture [13]

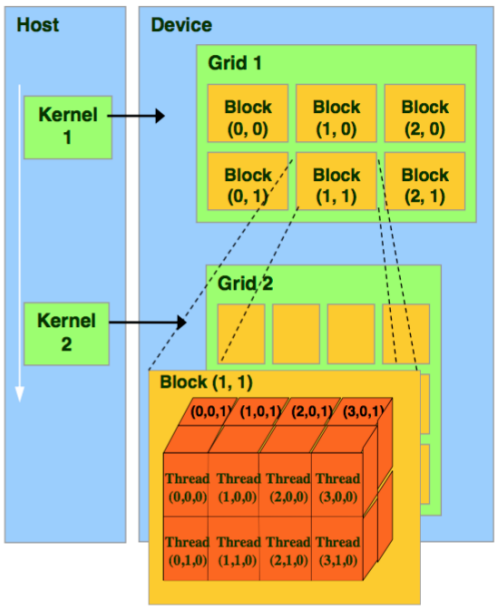

Figure 3.3: Cuda execution model [19]

cess the processing unit within that block.

3.2 Programming Models

As we said in the previous section, GPUs employe a SIMT model of execution, taking advantage of hundreds of cores, that can process thousands of software threads in parallel. These threads and also hardware cores are organized into two level of parallelism. Hardware cores are organized into an array of Streaming Multiprocessors (SMs) each SM consisting of a number of cores named as Scalar Processors(SPs). An execution of a computational kernel, e.g. CUDA kernel(more on this in the next section) will launch a software thread grid. As shown in the Fig.3.3,

Grid is the computational domain which consists of thread blocks which themeselves consist of threads. For exection of the kernels the CPU needs

to first move the input data to GPU DRAM through a PCIe or NVLink switch, then perform the computations within the kernel, finally move the results back to CPU. For programmers the challenge is to efficiently utilize the massive parallel capabilitites of GPUs, and to map the algorithms onto thread hierarchy and to lay out the data in both global memory and shared memory to maximize coalesced memory access for the threads and to avoid any pitfalls such as bank conflicts[20]. Using low level programming models such as CUDA and OpenCL is not only time consuming but also they need extensive change to the original source code, decreasing code readability. In this section we are going to talk about the most popular low level language based programming models for GPU, namely CUDA and then we are going to introduce the emerging standard in high level directive based GPU programming, OpenACC, which is becoming the standard in quick porting of scientific applications.

3.2.1 CUDA

The advent of multicore CPUs and manycore GPUs means that mainstream processor chips are now parallel systems. The challenge is to develope application software that transparently scales its parallelism to leverage the increasing number of processor cores. In November 2006 Nvidia introduced CUDA (Compute Unified Device Architecture), which is designed to overcome this challenge while maintaining a low learning curve for programmers familiar with standard programming languages such as C[6].
At it's core there are three key abstractions, a hierarchy of thread groups, shared memory 3.3 and barrier synchronization - that are exposed to porgammer as a set of language extensions.CUDA extends C by allowing the programmer to define functions, called *kernels*, that when called, are executed N times in parallel by N different CUDA threads, as opposed to only once like regular C functions.

A kernel is defined using the `__global__` declaration specifier and the number

of CUDA threads that execute that kernel for a given kernel call is specified using a `<<<..>>>` *execution configuration* syntax. Each thread that executes the kernel is given a unique *thread ID* that is accessible within the kernel through built-in variables. `threadIdx` is a 3-component vector, which themselves form a thread block, which in return form the grid for the current computational kernel, Fig.3.3
There is a limit to the number of threads per block, since all threads of a block are expected to reside on the same processor core and must share the limited memory resources of that core, the output of `deviceQuery.cu` is shown at appendix A for a Tesla V100 GPU on Marconi100 installation at CINECA.

On current GPUs, including Tesla V100 a thread block may contain up to 1024 threads. The kernel is actually executed in groups of 32 threads, called warp. Threads are executed by Scalar Processors(SPs), thread blocks are executed on multiprocessors, noting that thread blocks do not migrate and several concurrent thread blocks can reside on one multiprocessor, limited by the shared memory and registors of the multiprocessor, and finally a kernel is launched on a grid of thread blocks.

3.2.2 OpenACC

OpenACC is a GPU-based programming model that is emerging as the standard in directive based accelerator programming. As a joint standardization between CAPS, CRAY, PGI, and NVIDIA, OpenACC was first announced in 2011, which aims to provide a directive-based portable programming model for accelerators, By using OpenACC, it allows the users to maintain a single code base that is compatible with various compilers, while on the other hand the code is also portable across different architectures and in heterogenous platfroms. OpenACC is based on compiler directives that allow the developers to annotate parts of the source, that needs offloading to an accelerator, in a portable manner, without programmers prior extensive knowledge on the

the specifics of the accelerator being used, that can also scale to large counts of nodes at HPC centers. In C and C++, these directives take the form of a pragma. The example code below shows the OpenACC `kernels` directive without any additional clauses

```
1  #pragma acc kernels
```

In fortran, the directives take the form of a special comment, as demonstrated below,

```
1  !$acc kernels
```

as in this project, we mostly work with C source code, the code excerpts and examples, being discussed are in C, for the Fortran version the interested reader is encouraged to see [1].
OpenACC exposes programmer to three layers of parallelism, `vector` threads, which perform a single operation on multiple data(SIMD) in a single step, If there are fewer data than the length of the vector, operation is performed on null data and the results are discarded. A `worker` which computes one vector, and `gang` which is comprised of one or multiple `workers`. All `workers` within a `gang` can share resources, such as cache memory or processor. Mapping these high-level concepts to the loop constructs is vendor dependent, in a CUDA model these will be intepreted as *warps*, *threads*, and *thread blocks*, The `parallel` construct identifies a region of code that will be parallelized across OpenACC `gangs`, and when paired with the `loop` directive, the compiler will generate a parallel version of the loop for the accelerator. These two directives can, and most often are combined into a single `parallel loop` directive. Which can followed by the `private` or `reduction` caluses, much like `OpenMP` prgoramming model. These constructs can be seen in the snippet below for a jacobi relaxation method, which is ported with OpenACC directives,

```
while( err > tol && itr < MAX_ITR ) {

    err = 0.0;
    #pragma acc parallel loop reduction(max:err)
    for( int i = 1; i < N-1; i++ ) {
        #pragma acc loop reduction(max:err)
        for( int j = 1; j < M-1; j++ ) {

            A[i][j] = 0.25 * ( Anew[i][j+1] + Anew[i][j-1]
                             + Anew[i-1][j] + Anew[i+1][j] );
            err = fmax( err, fabs(A[i][j] - Anew[i][j]) );
        }
    }

    #pragma acc parallel loop
    for(int i = 1; i < N-1; i++) {
        #pragma acc loop
        for(int j = 1; j < M-1; j++) {
            A[i][j] = Anew[i][j];
        }
    }

    if( itr % 100 == 0 ) printf("%5d, %0.6f\n", itr, err);
    itr++;
}
```

Building the full jacobi iteration snippet with PGI compiler, gives:

```
$ pgcc -acc -ta:tesla:cc70 -Minfo=all jacobi.c
```

```
main:
     62, Generating create(Anew[:][:]) [if not already
         present]
         Generating copy(A[:][:]) [if not already present]
     67, Loop is parallelizable
         Generating implicit copy(error) [if not already
             present]
     69, Loop is parallelizable
         Generating Tesla code
         67, #pragma acc loop gang(32), vector(16) /*
             blockIdx.y threadIdx.y */
             Generating implicit reduction(max:error)
         69, #pragma acc loop gang(16), vector(32) /*
             blockIdx.x threadIdx.x */
     77, Loop is parallelizable
     79, Loop is parallelizable
         Generating Tesla code
         77, #pragma acc loop gang, vector(4) /* blockIdx.y
             threadIdx.y */
         79, #pragma acc loop gang(16), vector(32) /*
             blockIdx.x threadIdx.x */
     90, FMA (fused multiply-add) instruction(s) generated
```

with the `loop` directive we can give compiler more infomation about the proceeding loop, through several clauses, `independent`, which means that

all iterations of the loop are independent, which is implied by default by the PGI compiler, `collapse(N)`, which turns the next N loops into one flattened loop. `tile(N[,M,...])` which breaks the next 1 or more loops into tiles based on the provided dimensions, which uses data locality by reducing hits to the global memory. For optimizing data locality, OpenACC, provides also the `data` construct, which facilitates the sharing of data between multiple parallel regions. In the previous example we moved most of the compute intensive parts of the jacobi iteration to the accelerator, sometimes though process of copying data back and forth between host and device will consume more time than actual computation. When compilers fail to predict if and when the data will be needed, they sit on the safe side, and copy the data. Though in these cases programmers can help compiler, utilizing structured and unstructured *data* regions, moreover data clauses give the programmer additional control over how and when data is created on and copied to or from the device. These clauses may be added to any `data`, `parallel`, or `kernels` construct to inform the compiler of the data needs of that region of the code. Among others, the caluses that is used mostly in our code,

- `copy` - Allocate the listed variables on the device, copy to and from the device and then free the space on the device.
- `copyin` - Same as copy, without copying back the data.
- `copyout` - Allocates memory on the device without initializing them, at the end of the region, copy the results back to the host and frees the space on the device.
- `create` - Just allocates the space on device, but do not copy to or from the device.
- `present` - The listed variables are already present on the device.
- `deviceptr` - It tells the compiler that for the listed variables do not translate their addresses, as they use device memory which is managed

outside of OpenACC, this is used when OpenACC is mixed with other languages such as CUDA, as OpenACC is interoperable.

If we make use of this new data constructs, we can improve upon our earlier porting of jacobi iteration:

```
#pragma acc data copy(A) create(Anew)
while( err > tol && itr < MAX_ITR ) {

    err = 0.0;
    #pragma acc parallel loop gang(32),vector(16) reduction(
       max:err)
    for( int i = 1; i < N-1; i++ ) {
        #pragma acc loop reduction(max:err)
        for( int j = 1; j < M-1; j++ ) {

            A[i][j] = 0.25 * ( Anew[i][j+1] + Anew[i][j-1]
                             + Anew[i-1][j] + Anew[i+1][j] );
            err = fmax( err, fabs(A[i][j] - Anew[i][j]) );
        }
    }

    #pragma acc parallel loop
    for(int i = 1; i < N-1; i++) {
        #pragma acc loop gang(16),vector(32)
        for(int j = 1; j < M-1; j++) {
            A[i][j] = Anew[i][j];
        }
    }

    if( itr % 100 == 0 ) printf("%5d, %0.6f\n", itr, err);
    itr++;
}
```

On the other hand if we make use of object oriented programming, data is usually allocated in constructor and deallocated in the desctructor, and cannot be accessed outside of the class, or complex structures that make use of dynamically allocated data, in these cases structured data regions are not sufficient to tell the compiler the data layout. Hence OpenACC 2.0 introduced unstructured data lifetimes. The `enter data` directive with `create` and `copyin` data clauses may be used to specify how the data should be created on the device and `exit data` directive with `copyout` and `delete` clauses can be used to identify precisely when the data should be copied back and deallocated from the device.

`update` directive provides a way to synchronize content of host and device memory. it accepts a `device` clause for copying data from host to device and a `self` clause which was `host` in OpenACC 1.0 specification, but now is deprecated.

OpenACC Interoperability

OpenACC API specification is in a way that it doesn't force the programmer to choose OpenACC or some other accelerator programming model, like CUDA or OpenCL, it can play team with others, so that developers can choose OpenACC and other technologies. Here we mention some of these features that we have used in the LBE3D.
Suppose if you wanted to expose the device address of some array to host so that it can be passed to a function, the way to do it is with `host_data` region which accepts only the `use_device` clause.

```
void pbc_pop_soa_mpi_opt_packing(pop_type_soa *fp, pop_type*
    buffer_recv, pop_type* buffer_send) {

    //-----------------------------------------//
    //-----------------------------------------//
     if(size_comm > 1) {

         pack_pop_soa_X_up(fp, buffer_send);
#ifdef _OPENACC
#pragma acc host_data use_device(buffer_send, buffer_recv)
#endif
         MPI_Sendrecv(buffer_send, 5*NYP2*NZP2, MPI_DOUBLE,
            pxp, tag[0], buffer_recv, 5*NYP2*NZP2, MPI_DOUBLE,
             pxp, tag[0], MPI_COMM_ALONG_X, &status);
         unpack_pop_soa_X_up(fp, buffer_recv);

         pack_pop_soa_X_bottom(fp, buffer_send);
#ifdef _OPENACC
#pragma acc host_data use_device(buffer_send, buffer_recv)
#endif
         MPI_Sendrecv(buffer_send, 5*NYP2*NZP2, MPI_DOUBLE,
            pxp, tag[1], buffer_recv, 5*NYP2*NZP2, MPI_DOUBLE,
             pxp, tag[1], MPI_COMM_ALONG_X, &status);
         unpack_pop_soa_X_bottom(fp, buffer_recv);
     } else {
         //shared memory part...
     }
     // pbc in other directions
}
```

There may be an applicatin that has already been accelerated using languages such as CUDA or OpenCL, but the developer may want to add an accelerated region using OpenACC, in this case the API provides the `deviceptr` data clause, which may be used wherever any data clause may appear. This will inform the compiler that there is no need to do further action for that pointer. From CUDA 6.0, NVIDIA added support for Managed Memory,

which bridges the gap between host-device memory, resulting in simplifying programmers effort to manually manage data between host and device, which usually these two are seperated by PCI-Express bus. This is similar to OpenACC way of keeping referencs of the data, in that only one single reference to the memory is necessary and the runtime will handle the complexities of the data movement, such as in the case of C++ classes or structures with dynamically allocated array of pointers, which will normally require a manuall deep copy of all the elements.

Porting Cycle

The porting cycle usually is an incremental approach to insure correctness. Programmers usually assess the application performance, getting to know the hot spots of the application, then using OpenACC to parallelize important loops in the code, and then optimizing data locality to remove unnecessary data migrations between the host and the accelerator, and finally optimizing loops within the code to maximize performance on a given architecture. This approach has been successful in many applications because it prioritizes changes that are likely to provide the greatest returnes so that the programmers can quickly and productively achieve the acceleration.

OpenACC provides a fast way of porting scientific applictions, using a unified interface for all accelerators, avoiding locking down the application for a specific one. On the hand there is a tradeoff between performance and portability. Although OpenACC is desirable for it's simplicity and unified interface, but compared to, e.g. CUDA it's had a wrather significant performance dropoff, for a study of performance portability of OpenACC comapared to CUDA for LBM applications, see [5].

3.3 LBM based code with OpenACC

LBM codes, over the years, have been optimized for various architectures and HPC systems, including different CPU families, FPGAs and domain-specific machines. With the advancement of the GPU technologies, and their ever increasing computational powers and efficiency one would naturally develop LBM codes for these accelerators. But as it is usuall, efficient and complete LBM codes have usually large code base, and hence refactoring and rewriting, as we explained before, usually is not the best idea. This project builds on top of the work done on optimizations and performance analysis of the LBE3D, which is a 3D LBM CFD solver, built on top of a generic compiler/profiling library, *ftmake*[8, 9]. Study of this code is interesting not only because of the underlying physical applications, but as it consists of two main kernels, propagate and stream, which are respectfully memory bound and compute bound problems, makes it of interest to benchmark different accelerators and processors.
In here we will introduce the code, and then the porting of the code to multi-GPU paradigm and further optimizations are discussed.

3.3.1 Data Layout

With the advances of Many Integrated Cores (MIC) processors and continuous innovations in GPU architectures, applications that want to be performant, have to consider their data memory layout. This is even more relevant and rewarding for the two prominent kernels of LBE3D. The canonical data structure that would be natural to use with LBM stencil algorithms, is the Array of Structures (AoS), shown at Fig.3.4 with this schema all the

population data associated with one particular lattice index, are contiguous in memory, but the same lattice index of different populations are not, which results in non-unit strided access in memory, which haults the Single Instruction Multiple Data (SIMS) instruction sets. With the Structure

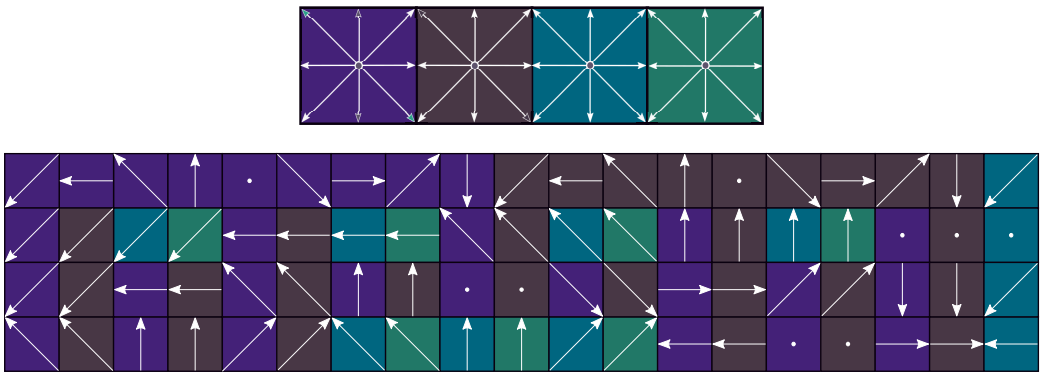

Figure 3.4: Top: lattice node for 4 population. Down: From top to bottom, *AoS*, *SoA*, *CSoA*, *CAoSoA*

of Arrays (SoA), distribution function for the same index populations for different lattice points are stored contiguous in memory, which is the appropriate layout to utilize SIMS and SIMT architectures. But as explained in [4] this will lead to unaligned memory access, since the address of popualtion is computed as the sum of the current populations plus an offset, which is generally not aligned with 64 Bytes, which will result in underutililzing the current `AVX512` instruction sets in e.g. Knights Landing (KNL) processors. There is another layout that is used in the LBE3D to reduce the amount of non-aligned accesses for differnet popualtion of a lattice site, which is to cluster sites of the lattice in one direction, e.g. in the case of a 2D lattice, with $L_x \times L_y$ sites, we cluster `VL` elements, which is a multiple of the vector size of the processor.

The propagate kernel job is to redistribute probabilty of each lattice site for the next iteration, Clustered Structure of Arrays (CSoA) will result in better performance in terms of bandwidth, but for the collision kernel, profiling have showed that this data structure will result in Translation Lookaside Buffer (TLB) misses[4]. We can, after partioning SoA into `VL` parts, cluster all ith elements of each partion into an array, resuting in the Clustered Array of Structure of Arrays (CAoSoA) data structure. This combines the benifit of the CSoA of having vectorizations of clusteres and at the same time, it clusters all population data of each lattice site at the aligned memory

addresses.

Here you can see the structure definitions and initializations, also you can see the `OpenACC` pragmas to allocate data on the device.

```
#define SIZE      (NX*NY*NZ)
#define SIZEP2    (NXP2*NYP2*NZP2)
#define SIZEPOVL (NXP2*NYP2*NZP2OVL)
typedef double poptype;
/*********************/
/* S T R C T U R E S */
/*********************/
typedef struct {
    poptype p[NPOP];
} pop_type;

typedef struct {
    poptype *f;
} pop_type_soa;

typedef struct {
    poptype c[VL];
} vpoptype;

typedef struct {
    vpoptype* p[NPOP];
} pop_type_csoa;

typedef struct {
    vpoptype p[NPOP];
} pop_type_caosoa;
/*************************/
/* A L L O C A T I O N S */
/*************************/
pop_type        *f_aos;
pop_type_soa     f_soa[NPOP];
pop_type_csoa   *f_csoa;
pop_type_caosoa f_caosoa;

posix_memalign((void*) &f_, SIZEP2 * sizeof(pop_type));
f_aos = f_;

/*********/
/* S o A */
/*********/
f_soa[0].f = f_
for(pp = 0; pp < NPOP; pp++) {
    f_soa[pp].f = f_soa[0].f + pp*SIZEP2;
}
#ifdef _OPENACC
#pragma acc enter data create(f_soa[NPOP])
for(pp = 0; pp < NPOP; pp++) {
#pragma acc enter data create(f_soa[pp].f[SIZEP2])
}
```

```
50 #endif
51
52 /***********/
53 /* C S o A */
54 /***********/
55 memset(f_, 0, NPOP*SIZEP2OVL, sizeof(vpoptype));
56 posix_memalign( (void*) &f_csoa,  DATA_ALIGNMENT, sizeof(
       pop_type_csoa));
57 #ifdef _OPENACC
58 #pragma acc enter data create(f_csoa)
59 #endif
60 f_csoa->p[0] = (vpoptype *) f_;
61 for(pp = 0; pp < NPOP; pp++) {
62     f_csoa->p[pp] = f_csoa->p[0] + (pp * SIZEP2OVL);
63 }
64 #ifdef _OPENACC
65 #pragma acc enter data create(f_csoa->p[0:NPOP][0:SIZEP2OVL])
66 #endif
```

Here also is the `move` kernel, which is used in the streaming step:

```
void move_soa(pop_type_soa * const __restrict__ nxt, const
   pop_type_soa * const __restrict__ prv) {
    int i, j, k;
    int p, idx, offset;

    profile_on ( __move_soa );

#if defined _OPENACC && !defined _OPENACC_OPT
#pragma acc update device(nxt[0:NPOP], prv[0:NPOP])
#pragma acc update device(nxt->f[0:SIZEP2], prv->f[0:SIZEP2])
#elif defined _OPENACC && defined _OPENCC_OPT
#pragma acc declare present(nxt, prv)
#pragma acc parallel loop collapse(3) private(i, j, k, idx,
   offset)
#else
#pragma omp parallel for private(i, j, k, p, idx, offset)
#endif
    for( i = 0; i < NX; i++ )
        for( j = 0; j < NY; j++ )
            for(p = 0; p < NPOP; p++)
                for( k = 0; k < NZ; k++ ) {

                    idx = IDX(i, j, k);
                    offset = idx + OFF[pp];
                    nxt[ p ].f[ idx ] = prv[ p ].f[ offset ];
                }

#if defined _OPENACC && !defined _OPENACC_OPT
#pragma acc update host(nxt->f[0:SIZEP2], prv->f[0:SIZEP2])
#pragma acc update host(nxt[0:NPOP], prv[0:NPOP])
#endif

    profile_off ( __move_soa__ );
}
```

and also the `collide` kernel, which is the compute bound kernel of the code and is used in the collision step:

```
1  void collide_soa (pop_type_soa * const restrict fp, const
       pop_type_soa * const restrict feq, double omega) {
2      int i, j, k;
3      int p, idx;
4
5      profile_on ( __collide_soa__ );
6
7  #if defined _OPENACC && !defined _OPENACC_OPT
8  #pragma acc update device(fp[0:NPOP], feq[0:NPOP])
9  #pragma acc update device(fp->f[0:SIZEP2], feq->f[0:SIZEP2])
10 #pragma acc parallel private(i, j, k, p, idx)
11 #elif defined _OPENACC && defined _OPENACC_OPT
12 #pragma acc declare present(fp, feq)
13 #pragma acc parallel private(i, j, k, p, idx) firstprivate(
       omega)
14 #else
15 #pragma omp parallel private(i, j, k, p, idx) firstprivate(
       omega)
16 #endif
17     for( p = 0; p < NPOP; p++ )
18 #ifdef _OPENACC
19 #pragma acc loop collapse(3)
20 #else
21 #pragma omp for collapse(3)
22 #endif
23         for( i = 0; i < NX; i++ )
24             for( j = 0; j < NY; j++ )
25                 for( k = 0; k < NZ; k++ ) {
26
27                     idx = IDX(i, j, k);
28                     fp[ p ].f[idx] = fp[ p ].f[idx] * ( 1 -
                           omega ) + omega * feq[ p ].f[ idx ];
29                 }
30
31 #if defined _OPENACC && !defined _OPENACC_OPT
32 #pragma acc update host(fp[0:NPOP])
33 #pragma acc update host(fp->f[0:SIZEP2])
34 #endif
35
36     profile_off ( __collide_soa__ );
37
38 }
```

3.3.2 Data Locality

The kernels of LBE3D are written separately so that benchmarking of different isolated parts can be done. But in general because of the memory wall, if the computational task can be done with less memory access, it is much more favourable, in this sense, we can fuse multiple kernels of the LBE3D into one, to avoid multiple read and write of the intermediate results and increase data locality.

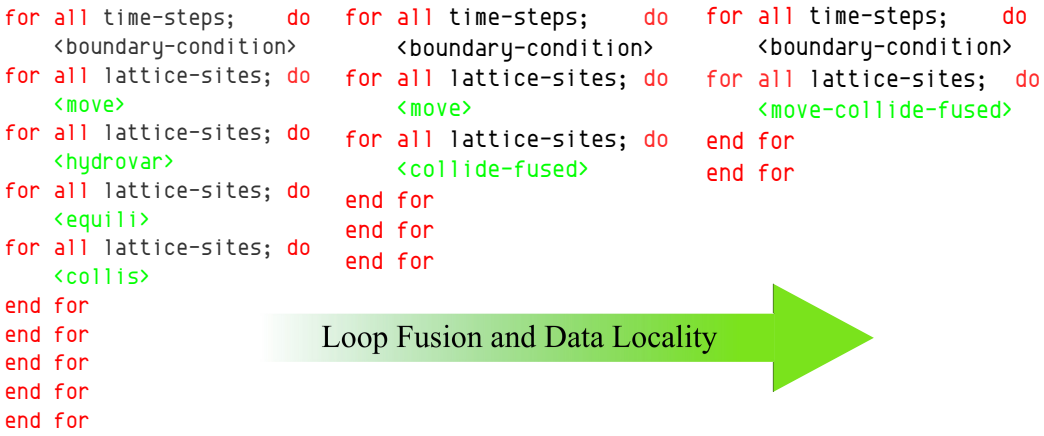

Figure 3.5: Figure showing different loop fusion levels and data locality abstraction schema. As we move forward we gain better and better performance as a result of enhanced data locality.

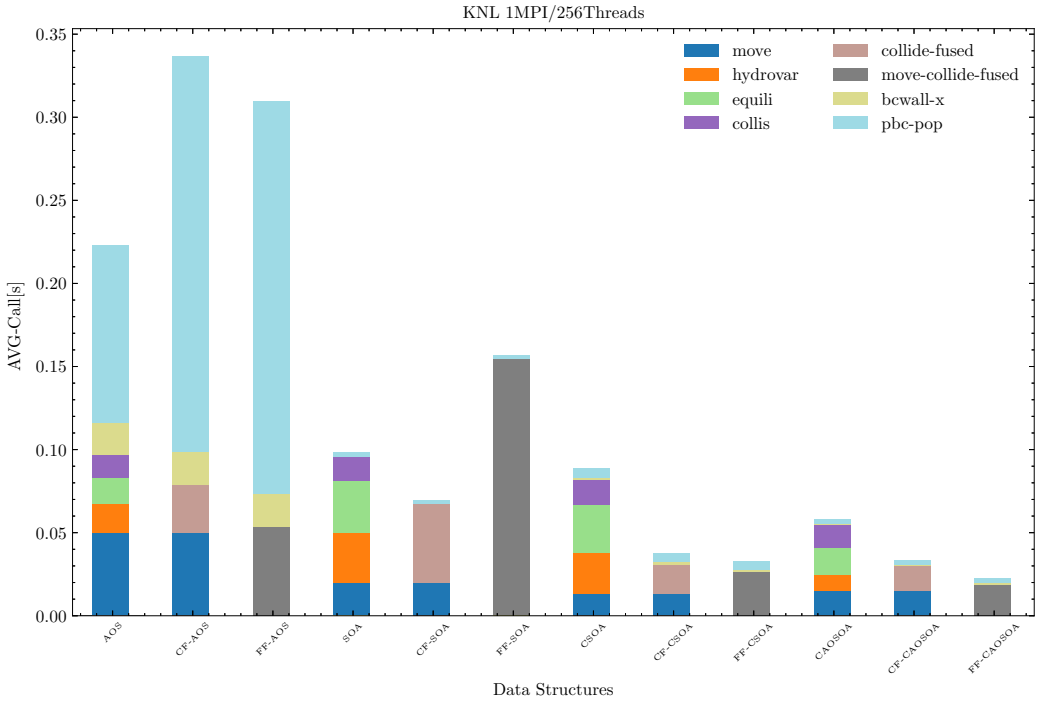

Figure 3.6: Average time per call for different data structures with 1 MPI processes and 256 Threads [9].

3.3.3 Porting and optimization of LBE3D with OpenACC

Schematic algorithm of the main LBE3D loop is shown below,

Algorithm 2: Schematics of the LBE3D Algorithm

```
Initialization;
Move lattice data to device;
for i ← 1 to NUM_STEPS do
    Set Boundary Condition;
    Stream populations for all lattice sites;
    Compute Hydrodynamical Quantities;
    Compute Equilibrium State;
    Collide populations for all lattice sites;
    if i%NUM_DIAG = 0 then
        Update lattice data on the host;
        Compute Hydrodynamical Quantities;
        Do Statistical Analysis ;
        Do Diagnostics ;
        Dump Data ;
    end
    Swap lattice pointers ;
end
```

We Start by allocating contiguous addresses on Graphics Processing Unit (GPUs) and controlling their affinity, as shown in the snippet. For a proof of concept, in a non-optimized version we copy all refrenced data in each kernel at each time step. Then we annotate the main loops over indices and populations with appropriate `loop` directive.

```
#ifdef OPENACC
    int ngpu = acc_get_num_devices(acc_device_nvidia);
    int igpu = me % ngpu;
    acc_set_device_num(igpu, acc_device_nvidia);
    if (AMIROOT) fprintf(stdout, "NUM GPU: %d\n", ngpu);
    fprintf(stdout, "GPU ID: %d, PID: %d\n", igpu, me);
```

```
#endif /*OPENACC*/
```

For scaling on multi-GPU the scaling of boundary conditions kernels is one of the determining factors. for that there are multiple optimized versions of the kernel that performs the boundary condition. For GPU-to-GPU communication we use CUDA-aware implementation of `spectrum-mpi`, which is IBM® production quality MPI implementation based on OpenMPI. For that we use the `#pargma acc host_data use_device(field)` which tells the compiler to map the GPU memory to host, so that it can be used in Message Passing Interface (MPI) calls. You can see in the snippet below for the optimized periodic boundary condition, that is sending and receiving the use case of this.

```
void pbc_pop_soa_mpi_opt_packing(pop_type_soa *fp, pop_type*
    buffer_recv, pop_type* buffer_send) {

    //-----------------------------------------//
    //-----------------------------------------//
     if(size_comm > 1) {

         pack_pop_soa_X_up(fp, buffer_send);
#ifdef _OPENACC
#pragma acc host_data use_device(buffer_send, buffer_recv)
#endif
         MPI_Sendrecv(buffer_send, 5*NYP2*NZP2, MPI_DOUBLE,
            pxp, tag[0], buffer_recv, 5*NYP2*NZP2, MPI_DOUBLE,
             pxp, tag[0], MPI_COMM_ALONG_X, &status);
         unpack_pop_soa_X_up(fp, buffer_recv);

         pack_pop_soa_X_bottom(fp, buffer_send);
#ifdef _OPENACC
#pragma acc host_data use_device(buffer_send, buffer_recv)
#endif
         MPI_Sendrecv(buffer_send, 5*NYP2*NZP2, MPI_DOUBLE,
            pxp, tag[1], buffer_recv, 5*NYP2*NZP2, MPI_DOUBLE,
             pxp, tag[1], MPI_COMM_ALONG_X, &status);
         unpack_pop_soa_X_bottom(fp, buffer_recv);
     } else {
         //shared memory part...
     }
     // pbc in other directions
}
```

For example one of the packing functions is shown in the next page,

```
void pack_pop_soa_X_up(const pop_type_soa * const restrict p,
    poptype * const restrict buffer_send)
{
    int cx_acc_p[5] = {1, 7, 8, 11, 13};
    size_t j, k, src, dest, ci;
    size_t count = 0;

#if defined __OPENACC && !defined __OPENACC_OPT
#pragma acc pcopyin(p[0:NPOP])
#pragma acc pcopyin(p->f[0:SIZEP2])
#pragma acc parallel loop collapse(3) private(src, dest,
   count)
#elif defined __OPENACC && defined __OPENACC_OPT
#pragma acc declare present(p, buffer_send)
#pragma acc parallel loop collapse(3) private(src, dest,
   count)
#else
#pragma omp parallel for
#endif /*OPENACC && OPENACC_OPT*/

    for ( ci = 0; ci < 5; ci++ ) {
        for( j = 0; j < NYP2; j++ ){
            for( k = 0; k < NZP2; k++ ){

                count = ci * NZP2 * NYP2;
                src   = IDX( NX, j, k );
                dest = count + ( j * NZP2 ) + k;

                buffer_send[ dest ] = p[ cx_acc_p[ci] ].f[
                    src ];
            }
        }
    }
}
```

Chapter 4

Results

Generally there are couple metrics of interest when studying the performance of the LBM code. Here we mainly focus on *Average Time per Call (AVGx-Call)*, which is the total time spent on the kernel divided by the number of calls to that kernel,

$$\mathsf{AVGxCall} = \frac{\text{Total Time}}{\text{Num Calls}}$$

Also another important figure is the *Million Lattice Updates per Second*, which is defined as,

$$\mathsf{MLUPs} = \frac{V.n}{t \times 10^6},$$

where V is the lattice size, n is the number of iterations, and t is the total execution time of the LBM code.

4.1 Poiseuille Flow

After completely porting the data structures for Poiseuille part, and doing the benchmark on 1 gpu of Marconi100, Out of all the data structures present, we soon realize that SoA would be efficient to focus porting efforts. Fig.4.1 shows the porting of all the data structures with OpenACC.

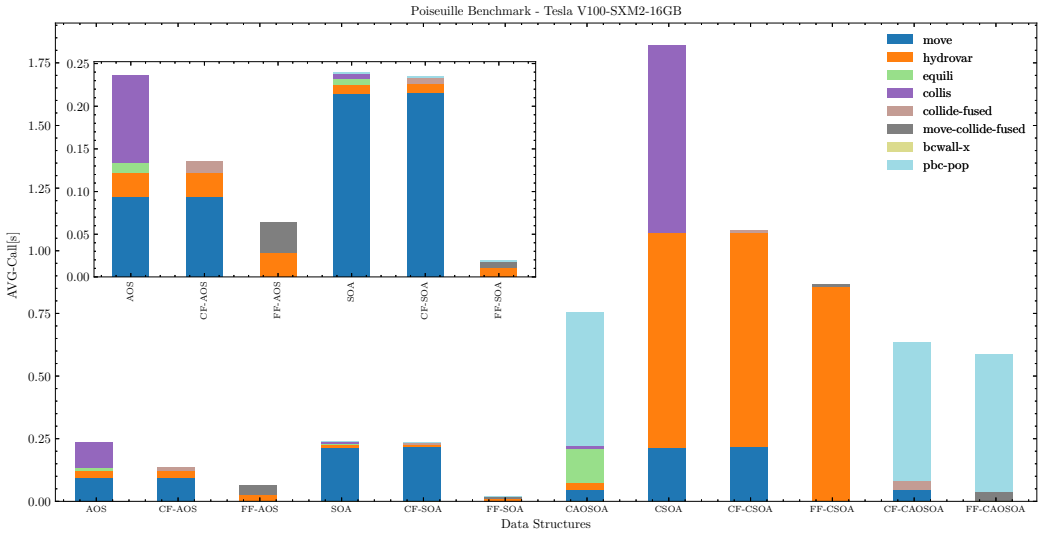

Figure 4.1: Average time per call with 1 MPI process for different levels of fusion with different data structures.

We soon reallize that the way to move forward would be to use Fully Fused (FF) version of SoA. After this initial test we move all the OpenACC data regions, namely data copyin and copyout pragmas out of the relevent kernels, which is declared with `OPENACC_OPT` identifier. By removing the cost of copy between host and device we move forward with the strong scaling test.

Fig.4.2 shows the strong scaling of the Poiseuille flow with 512 cubic lattice, with 2,...,16 nodes each with 4 Tesla V100 GPUs.

We see from the scaling figure that, between the three kernels, `pbc_pop`, doesn't scale past 8 nodes, and hence becomes a bottleneck in scaling. The reason for this bottleneck is the overhead of MPI communication for boundary exchange between halo cells of the lattice. To reduce the amount of calls,

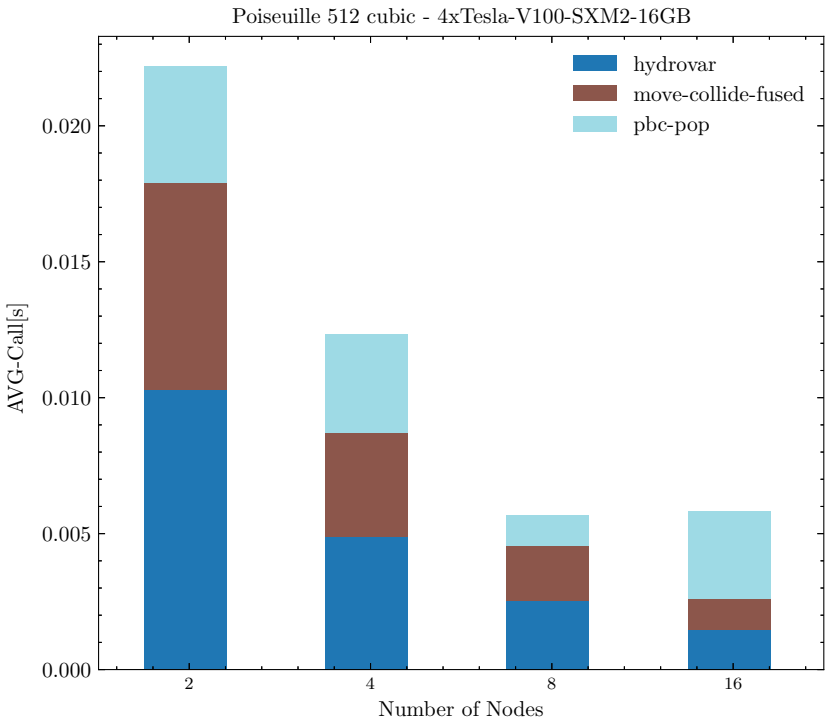

Figure 4.2: Strong scaling for Poiseuille flow with 512 cubic lattice.

before doing the `MPI_Sendrecv` calls, we pack one face of the lattice, for the populations that needes to be exchanged, which is 5, then we do the MPI exchange, and after that we unpack the buffer, see the snippet code provided for `pbc_pop_soa_mpi_opt_with_packing` kernel, in the previous section. Apart from the reduced number of calls, this has an also another advantage. For noncontiguous data layout in memory, MPI provides the `MPI_Datatype` that makes a contiguous type of the data, which essentialy stores the data in a buffer, here we have done the same thing, with `packing` functionality, but we have a advantage in that when we do a manual packing instead of using `MPI_Datatypes` we do the packing, multi-threaded using OpenACC or OpenMP which makes the code more performant.

Fig. 4.3 shows the speedup we achieved comparing `move_collide_fused_soa` average call time with 1 GPU of Tesla V100 and P100 with `move_collide_fused_csoa` kernel average time of single socket Intel Xeon processors. We see that mov-

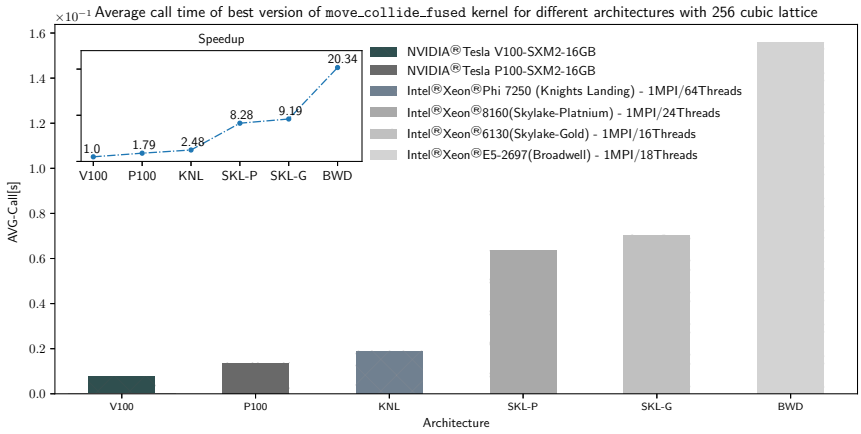

Figure 4.3: Speedup and Average call time for different architectures, with 256 cubic lattice. For CPU versions we have used the fully fused CSoA kernel data[9] and for GPU the fully fused version of SoA has been used.

ing from V100 to P100 we gain almost 2 fold increase in speed. Moving to CPU, KNL with 64 cores, is the best performant processor, among the Intel Xeon generation of MIC processors. This is because of it's core design goal, which is to directly compete with the likes of NVIDIA P100 in the HPC and Deep Learning domains. This data was gathered from the now dismantled KNL partition of MARCONI at CINECA. Compared with KNL, there is a 4 fold decrease in performance with SkyLake (SKL) processors. The Skylake Platnium is better than the Gold because of the extra 8 cores that it's packing. The SKL Platnium data is from the SKL partition of MARCONI and the SKL Gold is from the Marenostrum 3 supercomputer operated at Barcelona Supercomputing Center (BSC).

In Fig. 4.4 we present the figure for the effects of the $2d$ and $3d$ distribution of processes. We see that we have a slight performance gain using $2d$ distribution along xy coordinates. This is because of better mapping of MPI virtual cartesian topology, to the actual lattice problem. Since here we have a periodic boundary condition along the xy direction.

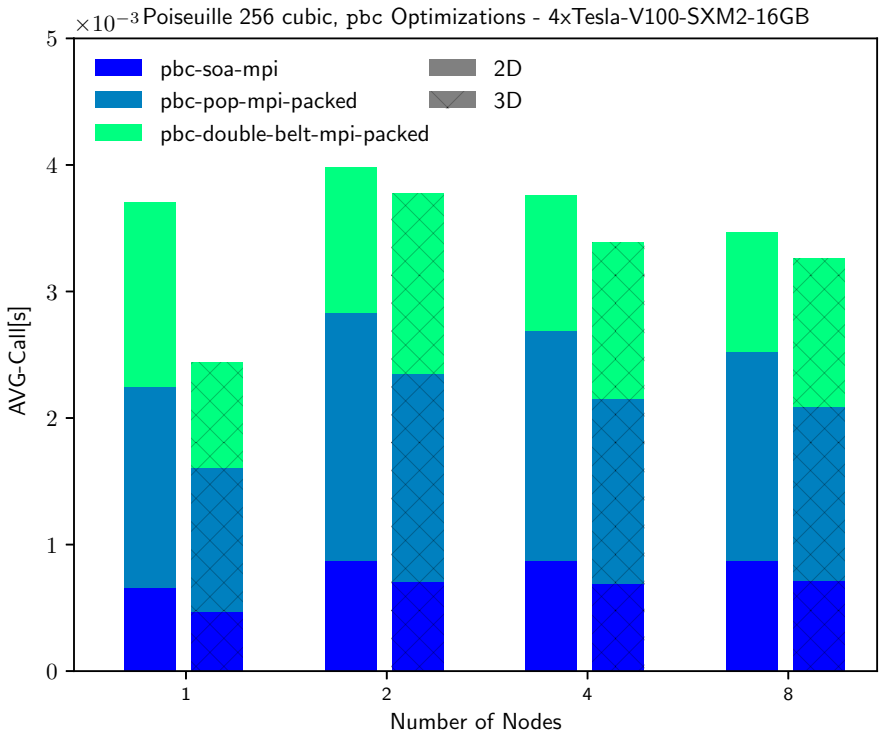

Figure 4.4: Effects of 2D and 3D distribution on boundary exchange average time. Notice the optimized time for 2D distribution.

4.2 Multi-Component

Poiseuille flow serves as a benchmark utility, so far major optimizations that we did was to reduce the data movement between host and device, by moving OpenACC data pragmas outside of the main loop. After that we implemented an optimized verison of `pbc_pop`, in that we create an MPI contiguous data type for the populations that needs to be exchanged between different GPUs, then we use a packing of those populations before doing the `MPI_Sendrecv`. We use these optimizations in the multi-component part. Fig. 4.5 shows the effects of the these optimizations. Also compared to the Poiseuille we have fused the `move` and `hydrovar` kernels, this results in significant speedup, as there will be enhanced data locality and cache efficiency.

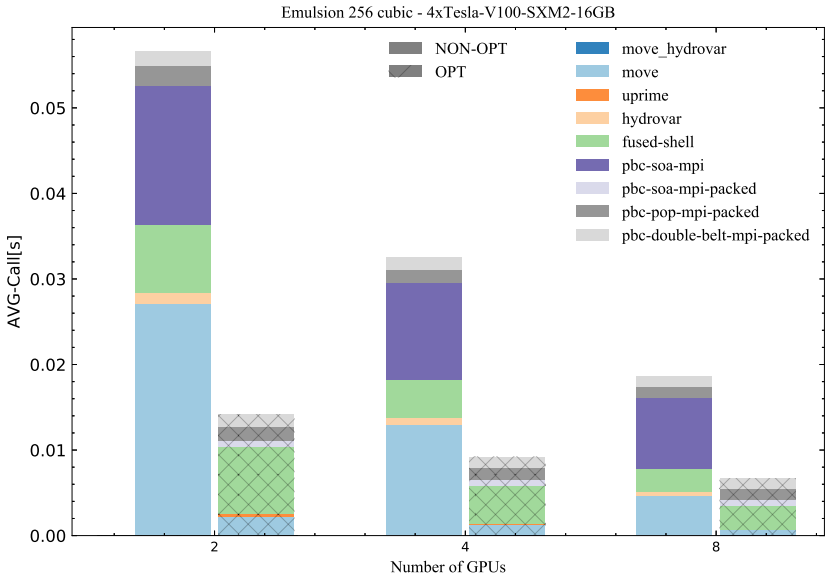

Figure 4.5: Effects of optmizations discussed in the text in the scaling. Observe the reduction in average time for boundary condition exchange and fusion of `move` and `hydrovar` kernels.

With the said optimizations we move forward with strong scaling benchmark for the multi-component. For that we use 256 cubic lattice for 1,...,64 nodes. Also shown is the efficiency for the scaling. We see that among others, `fused-shell` kernel hinders scaling, this kernel is used for introducing different forcing schemes in the fluid flow, and is the most compute intensive part of the multi-component part. The boundary condition exchange kernels remain almost constant during the whole scaling. compared with the `fused-shell`, `move-hyrovar` kernel, which is the fused versions of the `move` and `hydrovar`, has a better scaling figures.

In Fig. 4.7 we show the strong scaling with 512 cubic lattice, we see that the efficiency of scaling is better, since the bandwidth saturation and occupancy is higher, hence we expect to see even more efficient scaling with larger problem sizes.

Fig. 4.8 shows the weak scaling of multi-component part, with 2, 16, and 128 nodes with 256, 512, and 1024 cubic lattices respectively. From the efficiency

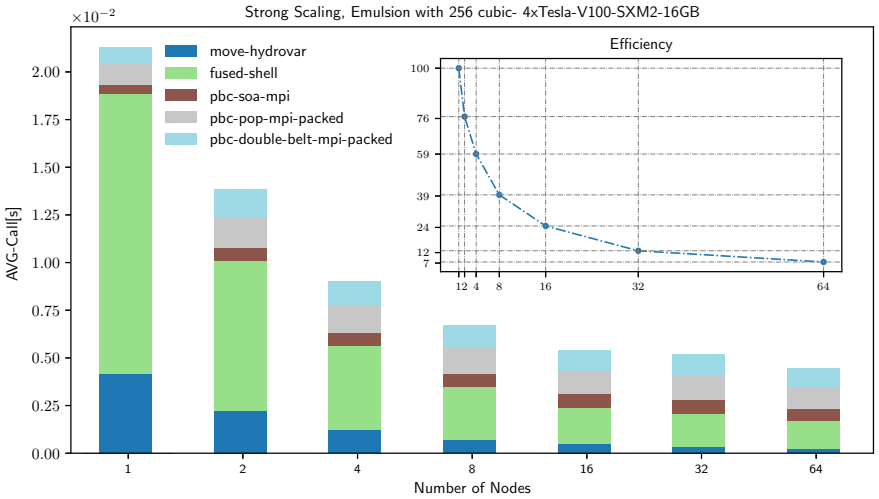

Figure 4.6: Strong scaling for 256 cubic lattice multi-component LBM.

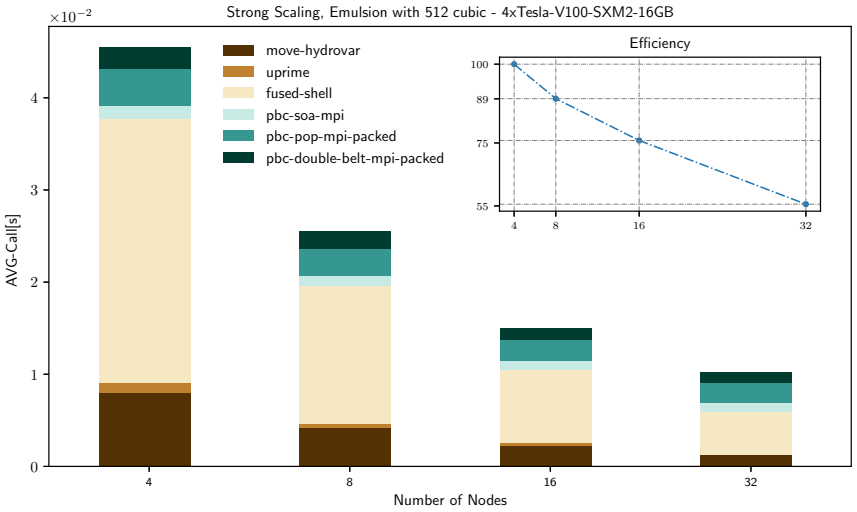

Figure 4.7: Strong scaling with 512 cubic lattice multi-component LBM.

graph we see that, we have a significant weak scaling efficiency in that we only drop 10% going from 2 to 128 nodes.

Complementing Fig. 4.3 we have the Fig. 4.9 that shows the average call time for V100 and SKL with 512 and 1024 cubic lattices. We can see that we have

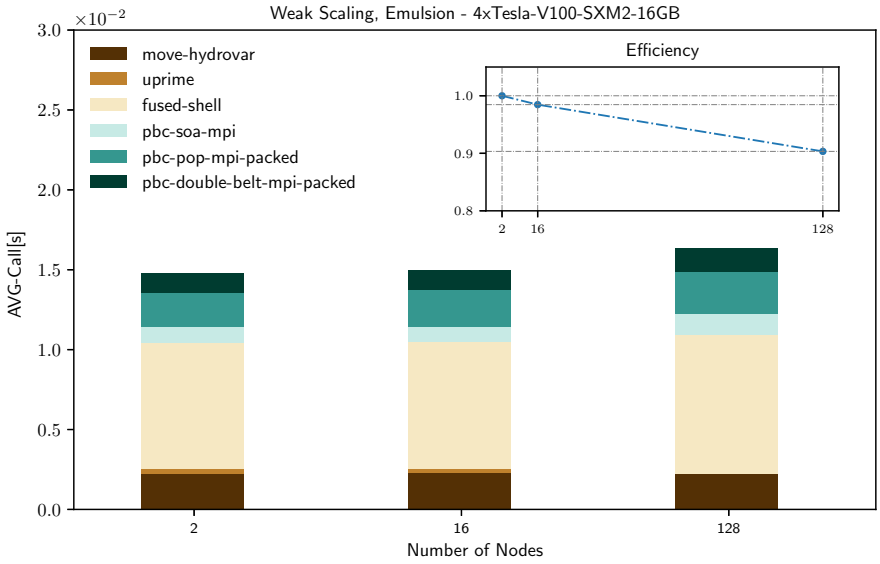

Figure 4.8: Weak scaling for multi-component with 2, 16, and 128 nodes with 256, 512, and 1024 cubic lattices.

almost equal total average time per call for each architecture but with V100 we need 8 times less node to achieve that time. This could be of interest in the sense of not only energy efficiency of executing on less nodes, which is something that could be studied further, but also on using compute hours on HPC centers.

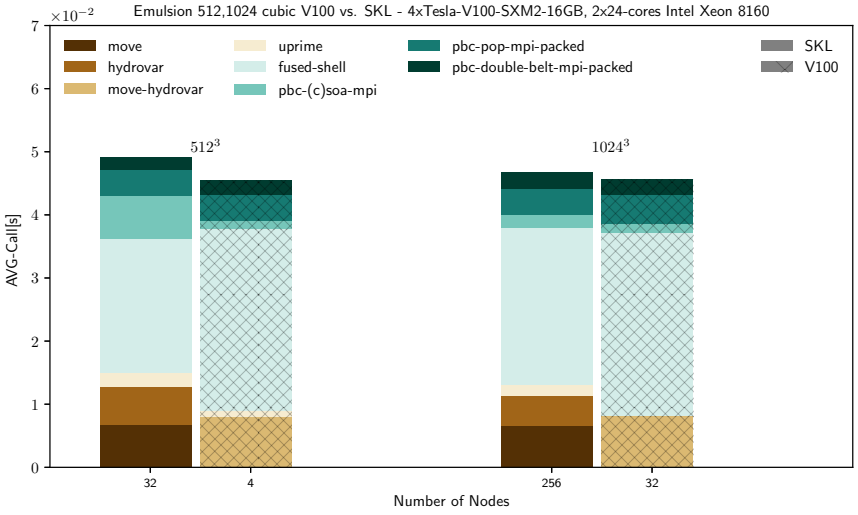

Figure 4.9: Comparison between V100 and SKL nodes of CINECA. The first group is with 512 cubic and the second group is with 1024 cubic lattices.

Chapter 5

Conclusions

In this project we have investigated an LBM application for it's portability to accelerators with directive based programming language, OpenACC. We have found that OpenACC provides a quick, unified and portable way of offloading the kernels of the LBE3D to GPUs. This will have performance degradation compared to native programming languages such as CUDA, but portability and simplicity of OpenACC makes it very interesting to investigate.

We started with porting the poiseuille benchamrk part of the application which is used to benchmark and validate the kernels. After single GPU porting we found out that the SoA data structure gives the best performance. This was given the fact that because of deep data structure with dynamic allocations, PGI compiler had an issue mapping the underlying pointers layout, which is something that needs to be investigated. After that we compare our best version with the previous data from the Intel®Xeon®family of processors, we find that compared to SKL processors we have about 8 times speedup. We find from strong scaling of poiseuille flow with 512 cubic lattice, that boundary condition exchange kernels, `pbc_pop` needs to be optimized for handling large amounts of calls for tens of nodes. For that we implement a packing versin of it, that uses a multi-threaded packing scheme to achieve an almost linear scaling.

After porting the poiseuille to multi-GPU, we begin the porting of the multi-component part. After fusing the `move` and `hydrovar` kernels, we implement the same packing schema for the double belt boundary condition exchange. After performing the strong scaling tests for 256 cubic lattice, we see that, past the 64 nodes the scaling efficiency drops drops, but on the other hand the weak scaling graph shows an almost constant scaling figure. By comparing the CPU and GPU results for multi-component part we see the same total average time with V100 nodes of MARCONI100 partition of CINECA, but with 8 times less nodes, which can be studied further for efficiency in energy consumption, also for quality of service and fare usage of HPC facilities with heterogenous architectures.

*References

[1] Openacc programming and best practices guide. Tech. rep., OpenACC Organization, 2015.

[2] Nvidia tesla v100 gpu architecture, the worlds most advanced data center gpu. Tech. rep., NVIDIA Corporation, 2017.

[3] Bhatnagar, P. L., Gross, E. P., and Krook, M. A model for collision processes in gases. i. small amplitude processes in charged and neutral one-component systems. *Phys. Rev. 94* (May 1954), 511–525.

[4] Calore, E., Gabbana, A., Fabio Schifano, S., and Tripiccione, R. Early Experience on Using Knights Landing Processors for Lattice Boltzmann Applications. *arXiv e-prints* (Apr. 2018), arXiv:1804.01918.

[5] Calore, E., Gabbana, A., Kraus, J., Schifano, S. F., and Tripiccione, R. Performance and portability of accelerated lattice boltzmann applications with openacc. *Concurrency and Computation: Practice and Experience 28*, 12, 3485–3502.

[6] Corporation, N. Cuda c++ programming guide. = https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html,, 2020.

[7] Dongarra, J., and Luszczek, P. *TOP500*. Springer US, Boston, MA, 2011, pp. 2055–2057.

[8] Girotto, I., Fabio Schifano, S., Calore, E., Di Staso, G., and Toschi, F. Performance and energy assessment of a lattice boltzmann method based application on the skylake processor. *Computation 8*, 44 (2020).

[9] Girotto, I., Schifano, S., Calore, E., Di Staso, G., and Toschi, F. Computational performances and energy efficiency assessment for a lattice boltzmann method on intel knl. *Advances in Parallel Computing 36* (2020), 605–613. cited By 1.

[10] Gunstensen, A. K., Rothman, D. H., Zaleski, S., and Zanetti, G. Lattice boltzmann model of immiscible fluids. *Phys. Rev. A 43* (Apr 1991), 4320–4327.

[11] Guo, Z., Zheng, C., and Shi, B. Discrete lattice effects on the forcing term in the lattice boltzmann method. *Phys. Rev. E 65* (Apr 2002), 046308.

[12] Hecht, M., and Harting, J. Implementation of on-site velocity boundary conditions for D3Q19 lattice Boltzmann simulations. *Journal of Statistical Mechanics: Theory and Experiment 2010*, 1 (Jan. 2010), 01018.

[13] Jia, Z., Maggioni, M., Staiger, B., and Scarpazza, D. P. Dissecting the NVIDIA Volta GPU Architecture via Microbenchmarking, Apr. 2018.

[14] Liu, H., Valocchi, A. J., and Kang, Q. Three-dimensional lattice boltzmann

model for immiscible two-phase flow simulations. *Phys. Rev. E 85* (Apr 2012), 046309.

[15] Luke Durant, Olivier Giroux, M. H., and Stam, N. Inside volta: The world's most advanced data center gpu. Tech. rep., NVIDIA Corporation, 2017.

[16] Shan, X., Yuan, X.-F., and Chen, H. Kinetic theory representation of hydrodynamics: a way beyond the Navier Stokes equation. *Journal of Fluid Mechanics 550* (Mar. 2006), 413–441.

[17] Shao, J. Y., Shu, C., Huang, H. B., and Chew, Y. T. Free-energy-based lattice boltzmann model for the simulation of multiphase flows with density contrast. *Phys. Rev. E 89* (Mar 2014), 033309.

[18] Succi, S. *The Lattice Boltzmann Equation for Fluid Dynamics and Beyond*. Clarendon Press, Oxford, 2001.

[19] Wolfe, M. Programming heterogeneous parallel architectures, directive programming with openacc. = https://www.pgroup.com/lit/presentations/cea-1.pdf, 2013.

[20] Xu, R. *Optimizing the Performance of Directive-Based Programming Model for GPGPUs*. PhD thesis, University of Houston, 2016.

[21] Yang, J. *Multi-scale simulation of multiphase multi-component flow in porous media using the Lattice Boltzmann Method*. PhD thesis, Imperial College London, 2013.

[22] Zhang, R., Shan, X., and Chen, H. Efficient kinetic method for fluid simulation beyond the navier-stokes equation. *Phys. Rev. E 74* (Oct 2006), 046703.

Appendices

Appendix A

Device Query

This is the output of the `deviceQuery.cu` for a node of MARCONI100 installation on CINECA supercomputing facility.

```
./a.out Starting...

 CUDA Device Query (Runtime API) version (CUDART static linking)

Detected 4 CUDA Capable device(s)

Device 0: "Tesla V100-SXM2-16GB"
  CUDA Driver Version / Runtime Version          10.2 / 10.2
  CUDA Capability Major/Minor version number:    7.0
  Total amount of global memory:                 16128 MBytes (16911433728 bytes)
  (80) Multiprocessors, ( 64) CUDA Cores/MP:     5120 CUDA Cores
  GPU Max Clock rate:                            1530 MHz (1.53 GHz)
  Memory Clock rate:                             877 Mhz
  Memory Bus Width:                              4096-bit
  L2 Cache Size:                                 6291456 bytes
  Maximum Texture Dimension Size (x,y,z)         1D=(131072), 2D=(131072, 65536), 3D=(16384, 16384, 16384)
  Maximum Layered 1D Texture Size, (num) layers  1D=(32768), 2048 layers
  Maximum Layered 2D Texture Size, (num) layers  2D=(32768, 32768), 2048 layers
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       49152 bytes
  Total shared memory per multiprocessor:        98304 bytes
  Total number of registers available per block: 65536
  Warp size:                                     32
  Maximum number of threads per multiprocessor:  2048
```

```
  Maximum number of threads per block:           1024
  Max dimension size of a thread block (x,y,z): (1024, 1024,
     64)
  Max dimension size of a grid size    (x,y,z): (2147483647,
     65535, 65535)
  Maximum memory pitch:                          2147483647
     bytes
  Texture alignment:                             512 bytes
  Concurrent copy and kernel execution:          Yes with 4
     copy engine(s)
  Run time limit on kernels:                     No
  Integrated GPU sharing Host Memory:            No
  Support host page-locked memory mapping:       Yes
  Alignment requirement for Surfaces:            Yes
  Device has ECC support:                        Enabled
  Device supports Unified Addressing (UVA):      Yes
  Device supports Managed Memory:                Yes
  Device supports Compute Preemption:            Yes
  Supports Cooperative Kernel Launch:            Yes
  Supports MultiDevice Co-op Kernel Launch:      Yes
  Device PCI Domain ID / Bus ID / location ID:   4 / 4 / 0
  Compute Mode:
     < Default (multiple host threads can use ::cudaSetDevice
        () with device simultaneously) >

Device 1: "Tesla V100-SXM2-16GB"
  .
  .
  .
Device 2: "Tesla V100-SXM2-16GB"
  .
  .
  .
Device 3: "Tesla V100-SXM2-16GB"
  .
  .
  .
  Compute Mode:
     < Default (multiple host threads can use ::cudaSetDevice
        () with device simultaneously) >
> Peer access from Tesla V100-SXM2-16GB (GPU0) -> Tesla V100-
   SXM2-16GB (GPU1) : Yes
> Peer access from Tesla V100-SXM2-16GB (GPU0) -> Tesla V100-
   SXM2-16GB (GPU2) : Yes
> Peer access from Tesla V100-SXM2-16GB (GPU0) -> Tesla V100-
   SXM2-16GB (GPU3) : Yes
> Peer access from Tesla V100-SXM2-16GB (GPU1) -> Tesla V100-
   SXM2-16GB (GPU0) : Yes
> Peer access from Tesla V100-SXM2-16GB (GPU1) -> Tesla V100-
   SXM2-16GB (GPU2) : Yes
> Peer access from Tesla V100-SXM2-16GB (GPU1) -> Tesla V100-
   SXM2-16GB (GPU3) : Yes
> Peer access from Tesla V100-SXM2-16GB (GPU2) -> Tesla V100-
   SXM2-16GB (GPU0) : Yes
> Peer access from Tesla V100-SXM2-16GB (GPU2) -> Tesla V100-
   SXM2-16GB (GPU1) : Yes
> Peer access from Tesla V100-SXM2-16GB (GPU2) -> Tesla V100-
   SXM2-16GB (GPU3) : Yes
> Peer access from Tesla V100-SXM2-16GB (GPU3) -> Tesla V100-
   SXM2-16GB (GPU0) : Yes
```

```
69 > Peer access from Tesla V100-SXM2-16GB (GPU3) -> Tesla V100-
       SXM2-16GB (GPU1) : Yes
70 > Peer access from Tesla V100-SXM2-16GB (GPU3) -> Tesla V100-
       SXM2-16GB (GPU2) : Yes
71
72 deviceQuery, CUDA Driver = CUDART, CUDA Driver Version =
       10.2, CUDA Runtime Version = 10.2, NumDevs = 4
73 Result = PASS
```